Two WAM Implementations of Action Rules

Bart Demoen[1] and Phuong-Lan Nguyen[2]

[1] Department of Computer Science, K.U.Leuven, Belgium
[2] Institut de Mathématiques Appliquées, UCO, Angers, France

Abstract. Two implementations of Action Rules are presented in the context of a WAM-like Prolog system: one follows a meta-call based approach, the other uses suspended WAM environments on the WAM heap. Both are based on a program transformation that clarifies the semantics of Action Rules. Their implementation is compared experimentally to the TOAM-based implementation of Action Rules in B-Prolog. The suspension based approach is faster at reactivating agents on the instantiation event. The meta-call approach is easier to implement, performs overall very good and much better for synchronous events, and it is more flexible than the suspension based approaches.

1 Introduction

The first publication of an implementation of delayed goals in the context of the WAM is by Carlsson [2]: a delayed goal is represented by a term on the heap and attached to a variable. The term is meta-called later. This method was originally only used for implementing *freeze/2*, and it has evolved into a more generally useful feature using attributed variables, in particular it is used in the implementation of (finite domain) constraint solvers.

In constraint solver programming, a constraint is often specified as a goal that waits to be re-executed every time one of the involved variables changes, e.g., an element of the domain is excluded, or the variable is fixed. It is important that the goal - usually a propagator - can be executed quickly, i.e., that the context switch from the normal execution to the propagator and back is cheap.

If the delayed goal needs to be executed on the instantiation of one variable, the term is meta-called just once. In other cases - e.g., when a domain change is the trigger - the goal possibly needs to be re-executed many times and the meta-call approach meta-calls the same term many times. Implementing this in the WAM is quite well understood and it requires no changes to the basic WAM architecture. However, in the WAM, meta-calling a term involves filling the argument registers, and most often, an environment for the called predicate must be allocated. Both add to the cost of the context switch.

In B-Prolog the cost of the context switch is kept down by exploiting the overall architecture of the TOAM [9,10]. Generally speaking, the TOAM pushes the execution state of predicates on the execution stack (including the information on alternative clauses) and passes arguments to calls on the same stack. Zhou used this mechanism for implementing *freeze/2* in [11]: for a delayed goal (named

M. Garcia de la Banda and E. Pontelli (Eds.): ICLP 2008, LNCS 5366, pp. 621–635, 2008.

an agent), the implementation builds a suspension frame on the execution stack, blocks it - i.e., protects it from being popped prematurely - and reuses it every time the agent is reactivated. In this way, the setup of the goal which the meta-call approach performs at every activation, is done only once. However, there are some disadvantages to blocking frames on the execution stack, the most prominent being that other frames can become unreachable while not on the top of the stack and that in the absence of backtracking, the space occupied by these frames cannot be recovered without a garbage collector for the execution stack.

In B-Prolog, suspended goals can be specified by Action Rules: the predecessors of Action Rules were named delay clauses in [11]. Action Rules offer two highly valuable features. First, with their powerful surface syntax, they allow a compact and concise specification of a goal waiting to be re-executed on different conditions. Secondly, Action Rules can be effectively mapped to efficient abstract machine code, at least in the TOAM. Indeed, the constraint solvers of B-Prolog derive their high performance partly from translating constraints to specialized Action Rules predicates [12]. The efficiency of the B-Prolog constraint solvers, as implemented with Action Rules, should be enough motivation to explore the implementation of Action Rules in any Prolog system. However, the perception exists that an efficient Action Rules implementation is reserved to the TOAM as implemented in B-Prolog. The challenge to WAM implementors is therefore clear: design an efficient WAM implementation of Action Rules, while not changing the WAM in a fundamental way.

Two designs for implementing Action Rules in the WAM look attractive: the first uses the meta-call approach to delaying goals and carefully applies a number of optimizations so that the desired performance is obtained. The second design uses suspension frames in the spirit of the TOAM, but in contrast with the B-Prolog approach, the suspension frames are kept on the WAM heap, not on the control stack. Similar optimizations are applied here as well. We have implemented these two approaches in hProlog (see [5] for the origin of hProlog). This allows us to compare these approaches experimentally in a meaningful way with each other, and with B-Prolog. The experience reported here shows that the TOAM does not have an inherent advantage over the WAM for implementing Action Rules.

We first introduce some Action Rules terminology in Section 2. Section 3 explains Action Rules by means of a program transformation to Prolog: such a transformation has not been described before. It is the starting point for an efficient meta-call based implementation of Action Rules. Section 4 describes the basics of how a WAM environment can be kept on the heap and used as a suspension frame for re-entering the same clause more than once. In Section 5 we use suspension frames on the WAM heap for implementing Action Rules, following a variant of the transformation in Section 3. Section 6 describes a number of implementation details. Section 7 contains an empirical evaluation and comparison of our implementations with B-Prolog. Section 8 argues why we prefer the meta-call approach. Section 9 concludes.

We assume working knowledge of Prolog [3], the WAM [1,7], the TOAM [10], and some acquaintance with Action Rules [12] and attributed variables (see for instance the documentation of [8]).

2 Action Rules Terminology

The words *event* and *agent* are overloaded in the original Action Rules terminology of [12]. Therefore, just for the sake of this paper, we will stick to the meanings described hereafter. One rule in Action Rules has the form:

```
Head, Guard, {EventPats} => Body.
```

The *Head* looks like the head of a Prolog clause: instead of full unification, it uses one-way unification, otherwise named matching. The *Guard* is a conjunction of guard goals. It functions like the guard in committed choice languages: once a guard succeeds, execution commits to that rule. [12] refers to the constituents between the {} as *event pattern*: *EventPats* is a comma separated list of such event patterns each of which can lead to reactivation of the agent. The *Body* looks like an ordinary Prolog clause body. We assume that the *Head* has distinct variables as arguments: one can move the head matching to the guard. B-Prolog puts restrictions on which guards are allowed, but such restrictions are not relevant for this paper.

The principal functor of the head is an Action Rules predicate symbol. An Action Rules predicate can be defined by more than one rule, but it cannot be the head of an ordinary Prolog clause at the same time. An *agent* corresponds to a call to an Action Rules predicate: it can be suspended and activated more than once.

3 How Action Rules Work

Informally, the meaning of an Action Rules predicate, is as follows: if *ins(X)* appears as the event pattern of the selected rule, the agent is reactivated when X is instantiated, or, said differently, when the *event* 'X becomes instantiated' occurs; if *event(X,M)* appears as the event pattern, the agent is reactivated every time there is a call *post_event(X,Mess)* and in the reactivated agent, M is replaced by Mess; if *generated* appears in the event pattern, the agent's body is executed immediately when the agent is created. We treat only these three event patterns explicitly in this paper, but extending our approach to other event patterns is straightforward. An agent dies when a rule without event patterns is selected.

This short description is not detailed enough for building a complete implementation of Action Rules, and lacking a formal Action Rules semantics, we have made a specification of the most important aspects of Action Rules by means of a program transformation to Prolog with attributed variables[1]. Our specification does not capture every aspect of Action Rules, let alone the full B-Prolog

[1] We use SWI-Prolog [8] syntax, but any variant will do.

behavior, but it makes the essentials of Action Rules easier to understand and it will be clear how to add the other features of the B-Prolog implementation. We start by showing the transformation on an example in Section 3.1. Section 3.2 describes the transformation in general, while Section 3.3 fills out the remaining details about events.

3.1 Transforming Action Rules to Prolog: An Example

Below is an Action Rules predicate *p/2* with three rules:

```
1  p(X,Y), G1, {ins(X), ins(Y), event(X,M)} => B1(X,Y,M).
2  p(X,Y), G2, {ins(Y), generated}          => B2(X,Y).
3  p(X,Y), G3                               => B3(X,Y).
```

in which `G` and `B` denote a guard and a body. We transform it to Prolog as follows:

```
4  p(X,Y) :- G1, !,
5            SuspGoal = p_agent(Message,Alive,X,Y),
6            register_events([ins(X), ins(Y), event(X,M)],SuspGoal).

7  p(X,Y) :- G2, !,
8            SuspGoal = p_agent(Message,Alive,X,Y),
9            register_events([ins(Y)],SuspGoal),
10           B2(X,Y).

11 p(X,Y) :- G3,
12           B3(X,Y).


13 p_agent(_,Alive,_,_) :- Alive == dead, !.

14 p_agent(M,_,X,Y) :- G1, !, B1(X,Y,M).
15 p_agent(_,_,X,Y) :- G2, !, B2(X,Y).
16 p_agent(_,Alive,X,Y) :- G3, Alive = dead, B3(X,Y).
```

The transformation generates two Prolog predicates: *p/2* and *p_agent/4*. The three clauses for *p/2* in lines 4..12 correspond to the three rules for *p/2* in lines 1..3. If the rule corresponding to the clause has event patterns, a term SuspGoal is created, and the call to *register_events/2* makes sure that this term is attached to the relevant variables as specified by the event patterns of the rule. The latter happens in lines 6 and 9. If the corresponding rule has no event patterns, or *generated* is one of its event patterns, the body is executed. This happens in lines 10 and 12.

The predicate *p_agent/4* has two extra arguments: the argument *Alive* represents the liveness of the agent; killing an agent is done by unifying this variable with the atom *dead*. The argument *Message* is a placeholder for the message sent in a *post_event(X,Message)* goal and thus corresponds to the second argument in an event pattern of the form *event(X,Message)*.

p_agent/4 is called when an event takes place that reactivates the agent. Its first clause checks the liveness of the agent: if the agent is dead already, then its reactivation just succeeds. The other clauses correspond to the rules of the Action Rules predicate: they check the guard, commit to a clause and execute the corresponding body. If the corresponding rule has no event patterns, the agent is killed, as in line 16. Note that this unifies the second argument of SuspGoal with the atom *dead*.

The two predicates *p/2* and *p_agent/4* correspond to two phases in the life of an agent. *p/2* is executed on the initial call and can register events depending on the event patterns of the selected rule: the agent is created, and then goes to sleep while waiting for events. *p_agent/4* is executed when the agent is reactivated: the event patterns are no longer needed. Reactivation of an agent happens by meta-calling the term, constructed as SuspGoal, as is explained in Section 3.3.

3.2 General Transformation from Action Rules to Prolog

The general transformation of an Action Rules predicate *p/n* is shown. Let

```
p(X1,...,Xn), Guard_i, {EventPats_i} => Body_i.
```

be the i^{th} rule. The transformation generates:

```
% code for p/n
% i-th clause corresponding to i-th rule
p(X1,...,Xn) :- Guard_i, !,
        SuspGoal = p_agent(Message,Alive,X1,...,Xn),
        register_events(EventPats_i,SuspGoal),
        exec_body(EventPats_i,Body_i).


% code for the suspended p_agent/(n+2)
% first clause
p_agent(_,Alive,_,...,_) :- Alive == dead, !.

% (i+1)-th clause corresponding to i-th rule
p_agent(Message,Alive,X1,...,Xn) :- Guard_i, !,
        kill(EventPats_i,Alive),
        Body_i.
```

Note that the arguments to *exec_body/2*, *kill/2* and *register_events/2* are manifest at transformation time, so their calls can be unfolded. We use {} for denoting

the absence of event patterns; syntactically, this is not accepted by B-Prolog. The definitions of *exec_body/2* and *kill/2* are:

```
exec_body(EventPats,Body) :-
        ((isin(generated,EventPats) ; EventPats == {}) ->
                Body
        ;
                true
        ).

kill(Es,Alive) :- Es == {} -> Alive = dead ; true.
```

3.3 Registering and Dealing with Events

The event pattern *generated* has no explicit post associated to it. We show the details for the two other event patterns: *ins/1* and *event/2*. Instantiation happens asynchronously, i.e., the Prolog unification routine intercepts the instantiation of a variable which has a goal waiting on its instantiation, and puts the goal in a queue. Goals from this queue are executed as early as possible. *Event/2* events happen by explicitly calling the predicate *post_event/2*.

Registering Events. For every event pattern in its first argument, the predicate *register_events/2* calls *register_event/2*[2]:

```
register_events([],_).
register_events([E|Es],G) :- register_event(E,G), register_events(Es,G).

register_event(ins(X),G) :- attach_goal(X,ins1,G).
register_event(event(X,_),G) :- attach_goal(X,event2,G).
register_event(generated,_).

attach_goal(X,E,G) :-
        (var(X) ->
                (get_attr(X,E,Gs) ->
                        put_attr(X,E,[G|Gs])
                ;
                        put_attr(X,E,[G])
                )
        ;
                true
        ).
```

attach_goal/3 builds a list of all the agents waiting on the same event.

[2] According to the B-Prolog manual, the *ins1* goal should be attached to all variables in the term X in *ins(X)*, but this does not affect the benchmarks.

Posting Events and Activating Agents. *Post_event/2* is implemented as:

```
post_event(X,Mes) :- get_attr(X,event2,Gs), !, send_message(Gs,Mes).
post_event(_,_).

send_message([],_).
send_message([G|Gs],Mes) :-
          send_message(Gs,Mes),
          G =.. [Name,_|Args],
          NewG =.. [Name,Mes|Args],
          call(NewG).
```

Posting a Herbrand event (corresponding to *ins/1*) consists in instantiating a variable. If X has an *ins1* attribute, and X is unified with a non-variable T, then *ins1:attr_unify_handler/2* is called with as first argument the *ins1* attribute of X and as second argument T. Remember that the *ins1* attribute is a list of goal terms. The handler is defined as[3]:

```
ins1:attr_unify_handler(Ins1AttrX,_) :- call_reverse_list(Ins1AttrX).

call_reverse_list([]).
call_reverse_list([G|Gs]) :- call_reverse_list(Gs), call(G).
```

Note that *call_reverse_list/1* and *send_message/2* are left-recursive: in this way, we respect the B-Prolog order of reactivating agents.

This concludes the transformation of Action Rules to Prolog: our implementations of Action Rules later on respect the implied semantics. Moreover, our meta-call approach is really an optimized version of the transformation.

4 Suspension Frames on the WAM Heap

We start with an example: below is a declaration and a clause for *foo/3*, one clause for *p/0* and a query with its resulting output.

```
:- suspension(foo/3).          p :-                          ?- p.
foo(X,Y,SuspTerm) :-               foo(X,Y,SuspTerm),           first(X,Y)
       writeln(first(X,Y)),        X = 1,                       next(1,Y)
       yield(SuspTerm),            resume(SuspTerm),            next(1,2)
       writeln(next(X,Y)),         Y = 2,
       leave.                      resume(SuspTerm).
```

The idea is that *yield/1* transfers control back to the caller and returns a description of an execution environment in its argument. The predicate *resume/1* uses this description to resume execution just after the call to *yield/1*. The predicate *leave/0* returns to the caller. With this informal explanation, the output

[3] The shown handler deals only with the case of unification of a variable with a non-variable: it can be extended easily to deal with the unification of two suspension variables.

from the query *?- p* can already be understood. A more detailed explanation follows.

The declaration *:- suspension(foo/3).* tells the compiler that the code for the (single) clause for *foo/3* must start with the instruction *alloc_heap*: it acts like the WAM instruction *allocate*, except that it allocates the environment - named a suspension frame - on the heap. No other changes to code generation are needed for *foo/3*. *Yield/1*, *resume/1* and *leave/0* are new built-in predicates.

The goal *yield(SuspTerm)* performs two actions:

- SuspTerm is unified with a *suspension term* with arity four. Its first two arguments are the current environment pointer, i.e., the pointer to the current suspension frame on the heap, and a code pointer that points just after the goal *yield(SuspTerm)*, i.e., the point at which execution can be resumed later; the other two arguments are used for holding a message and for indicating whether the term represents a live agent: this anticipates the use of suspension terms for implementing Action Rules.
- control returns to the caller of *foo/3* without deallocating the suspension frame.

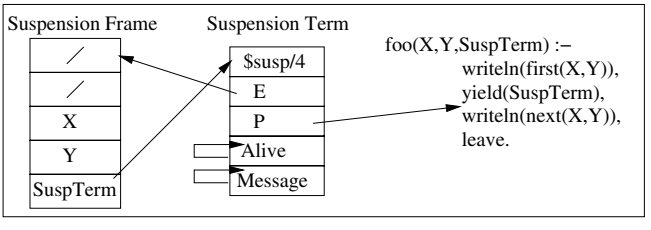

Fig. 1. Just after the execution of yield(SuspTerm)

The situation regarding the suspension frame and the suspension term (both on the heap) is depicted in Figure 1. The frame looks like an ordinary WAM environment, but its E and CP fields are irrelevant while the agent is sleeping, i.e., while no code in *foo/3* is executed.

The goal *resume(SuspTerm)* installs the environment pointer from SuspTerm in the WAM E register, and transfers control to the code pointed at by the code argument in the suspension term. It also fills out appropriately the E and CP fields in the suspension frame, for later use by *leave*. *Resume* can be called more than once with the same SuspTerm.

The goal *leave* returns to the caller of *foo/3* by using the E and CP fields in the current environment, which is in fact a suspension frame; *leave/0* does not deallocate the suspension frame.

The names *yield* and *resume* were chosen because of the obvious connection to coroutining. hProlog was extended with the new built-in predicates especially for our Action Rules experiment.

5 Using Heap Suspension Frames for Implementing Action Rules

This section is similar to Section 3: we start by redoing the example in Section 3.1, now using suspension frames on the heap. We skip Section 3.2 which generalizes the example: it should be clear how to do that. Section 5.2 is the heap suspension frame analogue of Section 3.3.

5.1 The Example

We reuse the example from Section 3.1. The transformation results in the following code for the two predicates *p/2* and *p_agent/3*:

```
15  p(X,Y) :- G1, !,
16            register_events([ins(X), ins(Y), event(X,M)],SuspTerm),
17            p_agent(X,Y,SuspTerm).

18  p(X,Y) :- G2, !,
19            register_events([ins(Y)],SuspTerm),
20            p_agent(X,Y,SuspTerm),
21            B2(X,Y).

22  p(X,Y) :- G3,
23            B3(X,Y).


24  :- suspension(p_agent/3).

25  p_agent(X,Y,SuspTerm) :-
26            yield(SuspTerm),
27            ( G1 -> pickup_message(SuspTerm,M), B1(X,Y,M), leave
28            ;
29              G2 -> B2(X,Y), leave
30            ;
31              G3, kill(SuspTerm), B3(X,Y), leave
32            ).
```

It should be clear how the three clauses in lines 15..23 correspond to the three rules in line 1..3. Also, the three disjunctive branches of *p_agent* in lines 27..31 correspond readily to those rules.

The goal *pickup_message(SuspTerm,M)* unifies variable *M* with the message slot in the suspension term SuspTerm: this slot is set by the predicate *set_message/2* that is explained in Section 5.2. The goal *kill(SuspTerm)* sets the live slot in the suspension term SuspTerm to *dead*: the built-in *resume/1* checks this slot before reactivating an agent. That is why *p_agent* does not check for liveness itself.

5.2 Registering and Dealing with Events

Registering Events. The code is the same as in Section 3.3, but now *attach_goal/3* builds a list of suspension terms.

Posting Events and Activating Agents. We need to redefine a number of predicates so that they take into account the fact that the attributes now contain a list of suspension terms. For the predicates dealing with the *ins1* event, these are:

```
ins1:attr_unify_handler(Ins1AttrX,_) :- resume_goals(Ins1AttrX).

resume_goals([]).
resume_goals([X|R]) :- resume_goals(R), resume(X).
```

Of the predicates dealing with the event2 event, only *send_message/2* needs adapting - *post_event/2* remains the same:

```
send_message([],_).
send_message([S|Ss],M) :-
     send_message(Ss,M),
     set_message(S,M),
     resume(S).
```

The idea is that at the reactivation of the agent the message is put in the *message* slot of the suspension term by the new built-in *set_message/2*. It is subsequently picked up in the body of the agent by *pickup_message/2*.

6 Making It Work

There are a few more issues to mention before the evaluation can take place.

The transformations. The transformations from Action Rules to Prolog described in Sections 3 and 5, have served as an explanation vehicle, and as the starting point for our implementations. However, as presented, the generated code still can benefit from some well understood optimizations: inlining, specialization, moving side-effect free code blocks ... Our final implementation applies such techniques. The most drastic change is that the two generated predicates *p* and *p_agent* for the suspension based method, are collapsed to one.

The representation of attributed variables. Our two implementations of Action Rules and the one in B-Prolog are based on some form of attributed variables. In B-Prolog those variables have several dedicated slots. We have applied that specialization to hProlog as well: for the purpose of this experiment, we have given hProlog attributed variables nine slots. The first slot is dedicated to the *ins/1* event pattern (used by *attach_goal(_,ins1,_)*) and the second to *event/2* (used by *attach_goal(_,event2,_)*). The seven remaining slots are meant for five different domain changes, a passive attribute and a finite domain: these are not used during the benchmarks, but they are properly initialized.

Low Level Support The predicates *yield/1*, *resume/1*, *leave/0* deal with the internals of the abstract machine, so they clearly must be implemented as low level built-ins. We have done the same with *pickup_message/2* and *set_message/2*. On top of that, some more effort was needed to achieve the desired performance:

- the last goal in a suspension predicate is *leave*; just before it, there is often a *call* instruction; a new instruction performs the action of both *leave* and *call*: the reason is mainly tail-call optimization.
- the code *G =.. [Name,_|Args], NewG =.. [Name,Mes|Args], call(NewG)* in Section 3.3 was collapsed to *event_call(Mes,G)*; *event_call/2* is one more new built-in predicate; in this case, a small implementation effort resulted in a large performance gain.
- the code implementing the meta-call moves the arguments of a term to the WAM argument registers; if *p* points just before the first argument of the term, then this code would routinely be written as:

```
for (i = 1; i =< arity; i++) Areg[i] = p[i];
```

  However, for performance it is better to unroll this, for instance to:

```
Areg[1] = p[1]; Areg[2] = p[2]; Areg[3] = p[3];
if (arity < 4) get_out_of_here;
Areg[4] = p[4]; Areg[5] = p[5]; Areg[6] = p[6];
...
```

  One should experiment to find the good unrolling granularity.
- we have also introduced a new instruction at the abstract machine level, that speeds up the reverse traversal of lists, as was needed in the predicates *call_reverse_list* and *send_message/2* (Sections 3.3 and 5.2).
- the general *attach_goal/3* predicate was specialized for its second argument to two built-ins *attach_ins1/2* and *attach_event2/2*.
- like some other implementations of the WAM, hProlog uses a separate stack for the choice points and the environments. With suspension frames on the heap, the *prev_E* field in an environment can point to the heap: code that maintains the top of environment stack TOS was adapted for this; moreover, the TOS is also pushed on the heap just before the suspension frame.

7 Evaluating the WAM Implementation of Action Rules

For the experiments, we have used B-Prolog 7.1b4.1 (the TOAM Jr. version [13]) and hProlog 2.9. The benchmarks were all run on a 1.8 GHz Pentium 4 CPU, under Debian. Garbage collection was avoided by starting the Prolog systems with enough initial memory. We always show timings relative to B-Prolog. B-Prolog is always at 100, and lower is faster. For the traditional benchmark set (not using Action Rules) hProlog 2.9 is about 10% faster than B-Prolog 7.1b4.1.

7.1 Original Benchmarks

Table 1 shows the results of running the benchmarks that were used in [11] to provide evidence for the qualities of the suspension mechanism in B-Prolog: it seems only fitting to use the same set here. These benchmarks only use the *ins/1* event pattern. The benchmarks are versions of the well known naive reverse, nqueens, sendmoremoney, and permutation sort, all adapted to use delayed goals: these benchmarks come with the B-Prolog distribution. In order to obtain meaningful timings, nrev was run on a list of length 500, nqueens computes all solutions for an 11x11 board, and the sort benchmark was given a list of 19 integers.

Table 1. The benchmark set of [11]

	bprolog	meta-call	suspension	# goals	# react
nrev	100	80	102	1	1
queens	100	88	111	10	4966
send	100	87	89	3	18412
sort	100	85	87	2	72827

Apart from the relative timings, Table 1 shows two characteristics of the benchmarks. Column *# goals* is the number of agents suspended on each variable, or equivalently, it is the length of the list built by *attach_goal* (for send, 3 is actually the maximal length and the average is 1.9). The last column shows the average number of times an agent is reactivated: the difference between nrev and the other benchmarks stems from the fact that only nrev is deterministic.

Table 1 shows that our implementation of suspension frames on the heap performs similar to the B-Prolog suspensions on the execution stack. It also shows that the meta-call approach performs very well.

The performance of the above benchmarks is not dominated enough by the operations related to delaying or waking goals. We therefore set up an artificial experiment that measures the operations in isolation as much as possible. The intention is to cancel out intrinsic performance differences between B-Prolog and hProlog as much as possible. This seems the best way to gain more insight in the relative performance of the operations we are really interested in.

7.2 Artificial Benchmarks for ins/1 and event/2

Table 2 summarizes the measurements on some artificial benchmarks. The B-Prolog agents have arity 7. Note that this means arity 9 for the term to be created and meta-called in the *meta-call/event2* entry . The benchmarks were implemented in B-Prolog with Action Rules, and by using their translation to our approaches. The columns represent the time needed to

- *freeze*: freezing a variable on a single goal; this measures agent creation.
- *melt*: melting a goal by instantiating a variable with a single goal suspended on it; this measures single agent reactivation on the *ins/1* event.

- *conjfreeze*: freezing one variable on 10^6 goals.
- *conjmelt*: melting a conjunction of 10^4 goals by instantiating a variable.
- *event2*: this corresponds to the cost of a goal *post_event(X,M)* when 10^4 agents are waiting on X to receive an *event/2* event.

Table 2. Some artificial benchmarks

	freeze	melt	conjfreeze	conjmelt	event2
bprolog	100	100	100	100	100
meta-call	70	87	56	86	44
suspension	97	57	78	58	101

The frozen goal was always of the form $p(\overline{X})$:- q, $r(\overline{X})$. (with trivial facts for *q/0* and *r/n*) so that in the meta-call approach an environment is allocated, and some argument saving/restoring is needed. Otherwise the meta-call approach would have been given an unfair advantage.

Table 2 shows that for both types of events, the meta-call approach is always faster than B-Prolog, and often significantly so. This seems incompatible with the idea that the TOAM has an inherent advantage over the WAM for suspending and reactivating agents. hProlog suspensions on the other hand perform almost equal to B-Prolog for *event/2* events, and hProlog is much faster for instantiation events. This shows that our implementation of suspension frames on the heap is of a decent quality.

hProlog is the first system to implement both a meta-call approach to Action Rules and a suspension frame approach. It is therefore interesting to see that the hProlog suspension approach performs better than the hProlog meta-call approach when goals are melted: this confirms the analysis of [11] experimentally.

8 Discussion

The outcome of the performance experiment does not make the choice between the two WAM approaches for implementing Action Rules easy: on one hand, the suspension based approach reacts faster to instantiation events, but the meta-call approach is much faster on sending messages. The latter is very common in constraint solvers, for all kinds of domain changes. Other considerations besides performance must be taken into account. Here is a short account of what we consider important.

- The memory foot-print is larger for the suspension approach than for the meta-call approach: one needs the suspension term and the suspension frame in the former case, and only the term to be meta-called in the latter case. We have measured total memory consumption[4] on some of the benchmarks of Section 7.1. B-Prolog uses between 15% more and 7% less memory than our suspension based method. The meta-call based method uses systematically 30% less than B-Prolog.

[4] The sum of the memory usage in the control stacks plus the heap.

- The suspension approach makes it more difficult to support recursive activation of agents, as for instance in the following rule:

```
p(X,Y), {ins(X), ins(Y)} => foo(X), Y = 2, bla(X,Y).
```

  On the query *?- p(X,Y), X = 1.* the goal $Y = 2$ reactivates the running agent. Also re-entering an Action Rules body through backtracking (which is not even supported in B-Prolog) is cumbersome and has a performance cost. In the meta-call approach, recursive activation of agents, as well as supporting backtracking into the Action Rules body, comes at no implementation or performance price.
- Clearly, the meta-call approach lends itself better to implementing custom tailored scheduling of agents: the agent is just a term which can be inspected and manipulated with the standard predicates. This is more difficult for suspension frames, whether on the heap or on the control stack.
- A dead agent is semantically garbage, but it can still be in the conjunction of agents attached to a variable: such a dead agent can be garbage collected, and B-Prolog does so. This can be done in both of our approaches. Neither approach seems to offer an advantage over the other on this issue.

Given the performance of the meta-call approach, its flexibility and its zero impact on the rest of the WAM implementation, we have a clear preference for the meta-call approach.

9 Conclusion

The basic suspension frame mechanism goes back to the first description of coroutines in [4]. It has been applied and reinvented many times. Environments on the WAM heap were used by Shen [6] in the DASWAM to implement and-parallelism: code executing with a heap environment can be suspended at any point, and resumed later from the same point once. In the case of Action Rules, execution can be resumed many times from the same suspension point. Those differences are not really important.

We are generally interested in understanding to what extent the TOAM gives a performance advantage over the WAM, and in this particular case for implementing Action Rules. Our results show that the WAM performs similar to the TOAM when a similar technique is used, namely suspension frames. Whether these frames are kept on the heap or on the control stack plays only a minor role, but in the WAM one would prefer the heap because that requires smaller changes to the abstract machine. However, it seems that a rather traditional meta-call approach to implementing Action Rules performs very good and often better. This is good news for WAM implementors, as a few small non-intrusive additions to the WAM suffice to achieve excellent performance. The choice for a meta-call approach to implementing Action Rules is justified further by the ease with which one can cater for recursive activation of agents, agents with a non-deterministic body, and custom build scheduling strategies.

[12] shows that Action Rules form a powerful tool for the constraint solver programmer. The efficient implementation of Action Rules seemed reserved to B-Prolog. This paper shows that also WAM based implementations can take advantage of the expressive power of Action Rules. Hopefully, this will have a positive impact on the future development of constraint solvers in WAM-based Prolog systems.

Acknowledgements. We thank Neng-Fa Zhou for helping us understand Action Rules. This work was partly done while the first author enjoyed the hospitality of the Institut de Mathématiques Appliquées of the Université Catholique de l'Ouest in Angers, France. We also thank Henk Vandecasteele letting us use his hipP compiler.

References

1. Aït-Kaci, H.: The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report (1990)
2. Carlsson, M.: Freeze, Indexing, and Other Implementation Issues in the WAM. In: Lassez, J.-L. (ed.) Logic Programming: Proc. of the Fourth International Conference, vol. 1, pp. 40–58. MIT Press, Cambridge (1987)
3. Clocksin, W., Mellish, C.: Programming in Prolog. Springer, Heidelberg (1984)
4. Conway, M.E.: Design of a Separable Transition-Diagram Compiler. Communications of the ACM 6(7), 396–408 (1963)
5. Demoen, B., Nguyen, P.-L.: So many WAM variations, so little time. In: Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Palamidessi, C., Pereira, L.M., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS, vol. 1861, pp. 1240–1254. Springer, Heidelberg (2000)
6. Shen, K.: Overview of DASWAM: Exploitation of Dependent And-parallelism. JLP 29(1-3), 245–293 (1996)
7. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Report 309, SRI (1983)
8. Wielemaker, J.: SWI-Prolog release 5.4.0 (2004), `http://www.swi-prolog.org/`
9. Zhou, N.-F.: Global optimizations in a Prolog compiler for the TOAM. Journal of Logic Programming 15(4), 275–294 (1993)
10. Zhou, N.-F.: On the Scheme of Passing Arguments in Stack Frames for Prolog. In: Proceedings of The International Conference on Logic Programming, pp. 159–174. MIT Press, Cambridge (1994)
11. Zhou, N.-F.: A Novel Implementation Method for Delay. In: Joint Internatinal Conference and Symposium on Logic Programming, pp. 97–111. MIT Press, Cambridge (1996)
12. Zhou, N.-F.: Programming Finite-Domain Constraint Propagators in Action Rules. Theory and Practice of Logic Programming (TPLP) 6(5), 483–508 (2006)
13. Zhou, N.-F.: A Register-Free Abstract Prolog Machine with Jumbo Instructions. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 455–457. Springer, Heidelberg (2007)